第8章 软件架构设计和实现

王璐璐 wanglulu@seu.edu.cn
廖力 lliao@seu.edu.cn

第8章 软件架构设计和实现

- 软件架构是软件系统质量的核心
- 成功的软件架构应具有以下品质
  - （1）良好的模块化
  - （2）适应功能需求的变化，适应技术的变化
  - （3）对系统的动态运行有良好的规划
  - （4）对数据的良好规划
  - （5）明确、灵活的部署规划

第8章 软件架构设计和实现

8.1 从需求分析到架构设计
8.2 从软件架构到详细设计
8.3 架构设计原则
8.4 软件架构设计面临的主要威胁及其对策

8.1 从需求分析到架构设计

- 需求分析和软件架构设计是软件全生命周期中两个关键活动。
- 需求分析主要考虑问题域和系统责任，其目的是得到一个正确、一致并且无二义的需求规约，作为后续开发、验证及系统演化的基础。
- 软件架构设计是一个架构的定义、文档编写、维护、改进和验证正确实现的活动。它的目的是将系统的高层结构显式地表达出来，在较高的抽象层次上为后续的开发活动提供“蓝图”。

8.1 从需求分析到架构设计

- 需求到软件架构的映射是一个既复杂又细致的工作，对相同的需求采用不同的映射机制会得到不同风格的架构。
- Paul Grunbacher认为有下面两个概念能够有效指导如何从需求向架构进行映射
  - （1）架构风格（architecture style）：它可以捕获结构、行为和交互模式等；
  - （2）特定领域的软件架构（DSSA, domain-specific software architecture）：一个DSSA关联一个模型和它的应用领域。

8.1 从需求分析到架构设计

- 在从用户需求获取软件架构的过程中，有两件事情必须要做的：
- （1）探索如何用软件架构的概念和描述手段在较高抽象层次上刻画问题空间（用户需求）的软件需求，获得软件需求规约说明书；
- （2）探讨如何从软件需求规约说明书自动或半自动地变换到软件架构设计。

8.1.1软件架构对需求的影响

- 用传统的方法产生需求规约，不考虑软件架构概念和原则，则在软件架构设计阶段建立需求规约与架构的映射将相对困难。
- 若把架构概念引入需求分析阶段，有助于保证需求规约、系统设计之间的可追踪性和一致性，有效保持软件质量。
- 将软件架构概念和原则引入需求分析，也可以让我们获得更有结构性和可重用的需求规约。

8.1.1软件架构对需求的影响

- 梅宏等提出的一种面向软件架构的需求工程方法步骤
  - （1）确定问题空间范围。
  - （2）确定系统用户及他们的职责。
  - （3）从用户角度探索整体外部行为，产生系统的顶层模型；
  - （4）确定组件和连接件 。
  - （5）规约组件和连接件。
  - （6）建立面向软件架构的高层需求规约。所得到的需求规约包括组件的需求规约、连接件的需求规约及约束需求规约等。
  - （7）检查面向软件架构需求规约，包括完整性、一致性和死锁等。

8.1.2基于软件需求的软件架构设计

- 从软件需求到软件架构存在的难点
  - （1）软件需求是频繁获取的非正规的自然语言，而软件架构规约常常是一种正式的语言。
  - （2）系统属性中描述的非功能性需求通常很难在架构模型中形成规约。
  - （3）需要以迭代和同步演化的方式进行软件需求理解和软件架构开发。
  - （4）在从软件需求映射到软件架构的过程中，保持一致性和可追溯性很难，且复杂程度很高。
  - （5）大规模系统必须满足数以千计的需求，从而导致很难确定和细化包含这些需求的架构相关信息。
  - （6）软件需求和软件架构需要满足不同的利益相关者的需求，很难在这些不同利益中找正确的平衡点。

8.1.2基于软件需求的软件架构设计

- 把软件需求模型转换为软件架构模型的代表性方法：

（1）面向目标和场景的软件架构设计方法

（2）基于APL（architecture prescription language）软件架构设计方法

（3）基于规则的架构决策诱导框架的软件架构设计方法

（4）基于全局分析的架构形成方法

（5）面向模式的架构形成方法

（6）需求与架构协同演化方法

（7）基于CBSP的架构形成方法

（8）基于WinWin模型与CBSP结合的架构形成方法

（9）面向特征的映射和转换方法

（I）面向目标和场景的软件架构设计方法

- Lin Liu等提出了一种面向目标的需求语言GRL(Goal-oriented Requirement Language)和一种面向场景的架构符号UCM(Use Case Maps)，支持面向目标和代理的建模和论证，并指导架构设计过程。
  - GRL是一种支持目标和代理目标建模、推导需求的语言，特别是处理非功能需求。
  - UCM提供一种场景可视化符号，描述和推导系统中大粒度行为模式。
- 该建模方法旨在对需求进行启发、求精和实现，直到得到一个满意的架构设计方案。

（I）面向目标和场景的软件架构设计方法

- 步骤：
  - （I）建立面向目标的需求模型。
  - （2）建立用例图模型。
  - （3）在用例图中划分部件，得到一个参考的架构设计方案。
  - （4）在需求模型中对目标进行求精细化并评估其非功能目标，如果满足则转入步骤6。
  - （5）若设计无法满足系统的非功能性目标，则分析找出新的目标和需求，寻求其他的解决方案。再返回步骤4
  - （6）考虑其他的可行方案，再利用系统的非功能目标进行权衡，找出最佳的架构设计方案。

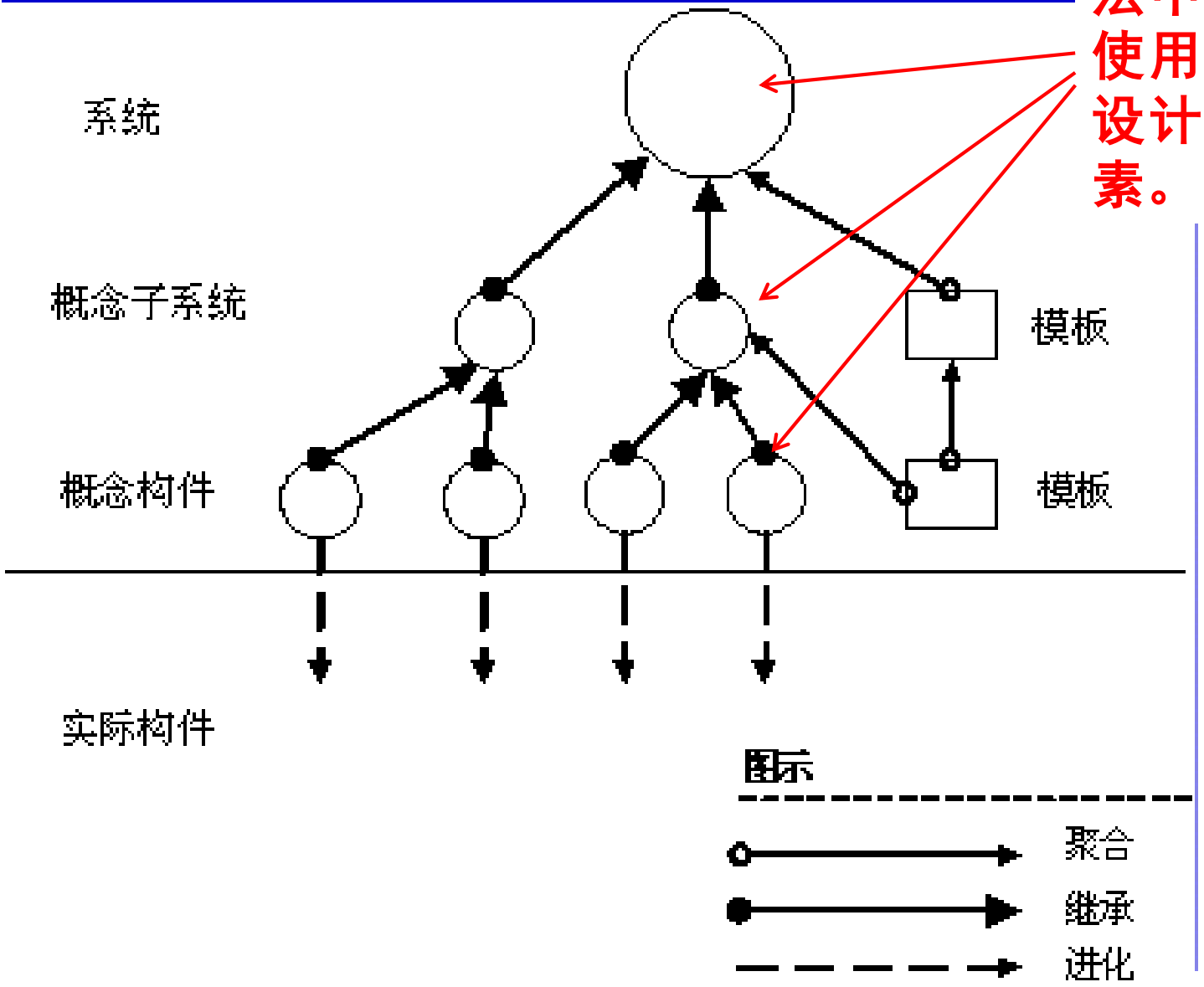
基于体系结构的软件设计方法

- 基于体系结构的软件设计（architecture-based software design, ABSD）方法为软件系统的概念体系结构提供构造方法，概念体系结构描述了系统的主要设计元素及其关系。

- 概念体系结构代表了在开发过程中作出的第一个选择，它是达到系统质量和业务目标的关键，为达到预定功能提供了基础。

基于体系结构的软件设计方法

ABSD方法中所使用的设计元素。



基于体系结构的软件设计方法

- ABSD方法有三个基础：

(1) 功能分解：在功能分解中，ABSD方法使用已有的基于模块的内聚和耦合技术；

(2) 通过选择体系结构风格来实现质量和业务需求。

(3) 软件模板的使用：利用一些软件系统的结构。

基于体系结构的软件设计方法

- 软件模板是一个特殊类型的软件元素，包括描述所有这种类型的元素在共享服务和底层构造的基础上如何进行交互。

- 软件模板还包括属于这种类型的所有元素的功能，这些功能的例子有：每个元素必须记录某些重大事件，每个元素必须为运行期间的外部诊断提供测试点等。

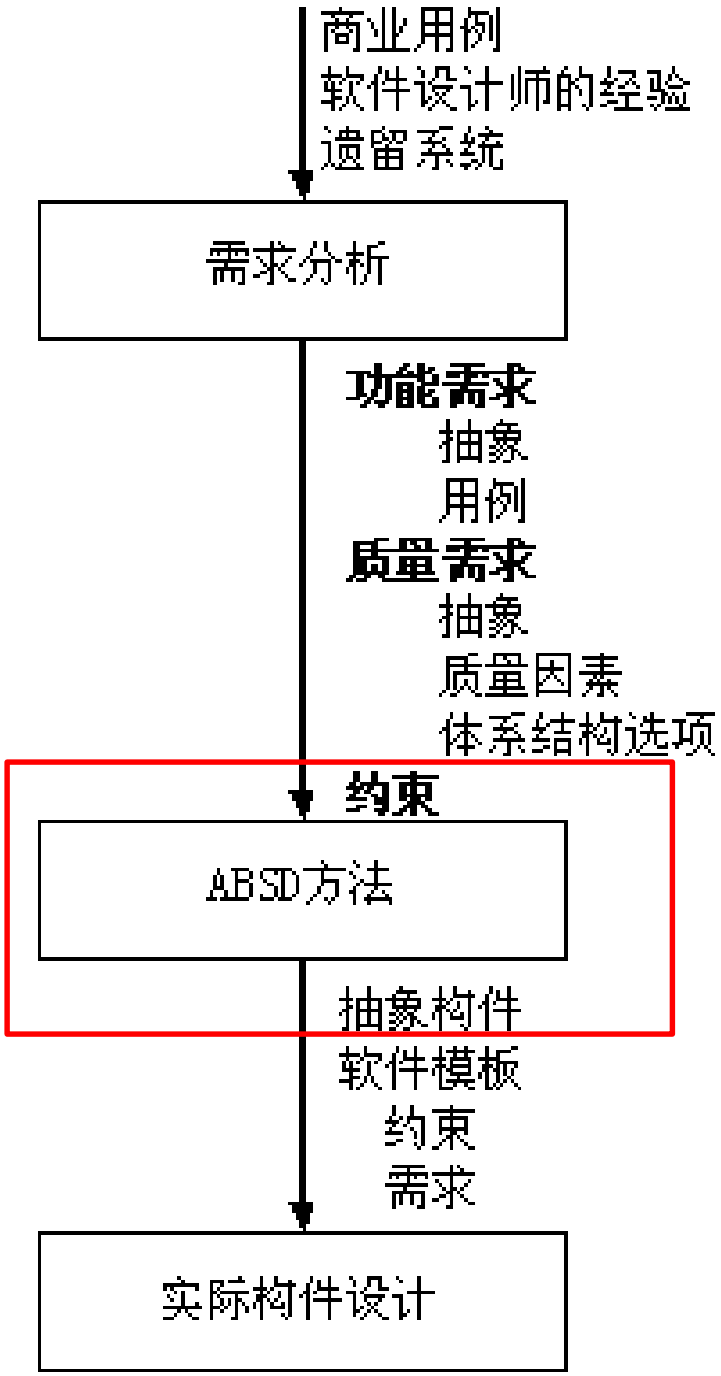
基于体系结构的软件设计方法

- ABSD方法的目的是组织最早的设计策略，不包括形成实际的软件构件和类。但要作出有关功能划分和达到不同质量属性机制的决策。

- ABSD方法是一个递归细化的方法，软件系统的体系结构通过该方法得到细化，直到能产生软件构件和类。

ABSD方法与生命周期

ABSD方法在生命周期中的位置



ABSD方法与生命周期

ABSD方法的输入由下列部分组成：

- **抽象功能需求**：功能需求的抽象描述，及其粗略变化。
- **用例**：用户与系统之间交互的具体表述,仅考虑重要的用例。
- **抽象的质量和商业需求**：质量需求尽量具体化。
- **质量因素**：实际质量和商业需求。采用质量场景可以对质量需求进行特定扩充，使质量因素具体化。
- **体系结构选项**：对于每个质量和业务需求，都要例举出能满足该需求的所有可能的体系结构。
- **约束**：前置的设计决策。例如：必须考虑某遗留系统特征。

ABSD方法的步骤

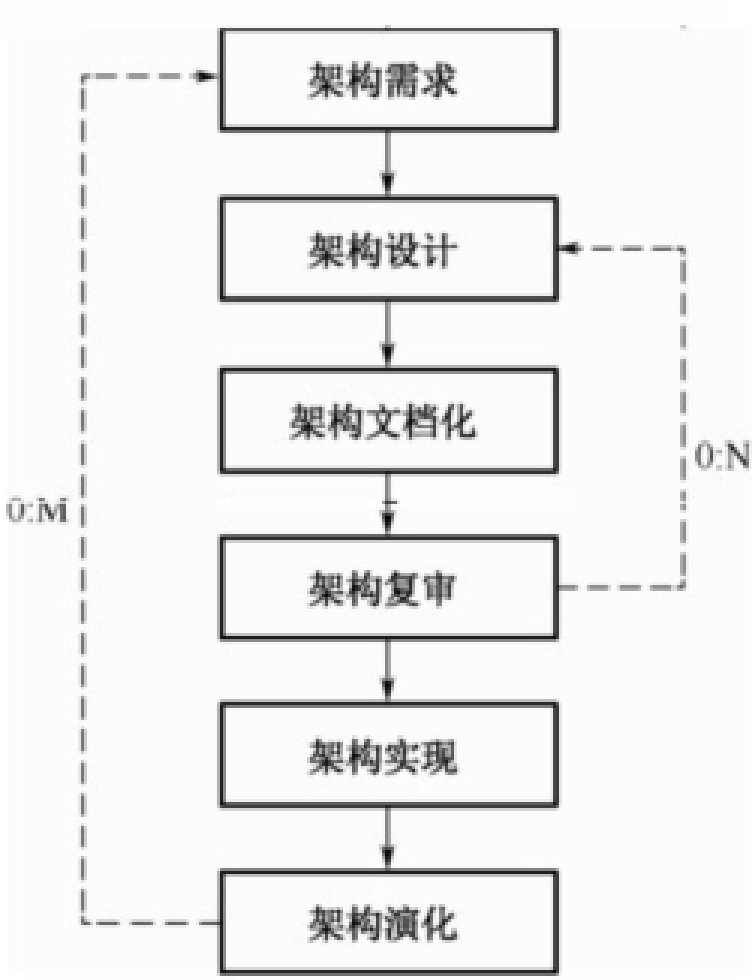


图　　基于架构的软件开发模型

ABSD方法的步骤

（1）架构需求

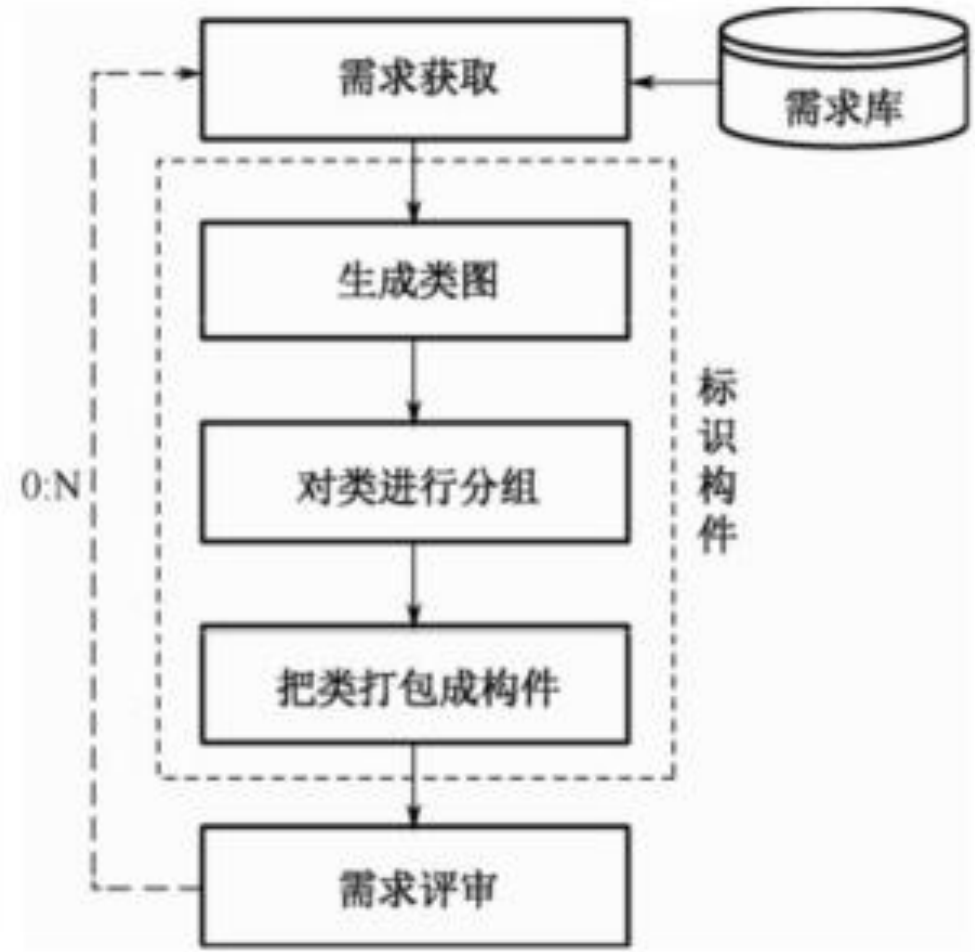


图　　架构需求过程

ABSD方法的步骤

（1）架构需求——功能分解

一个设计元素有一组功能，这些功能必须分组。分解的目的是使每个组在体系结构内代表独立的元素。分解可以进一步细化。

功能的分组可选择几个标准：

1）功能聚合。

2）数据或计算行为的类似模式。

3）类似的抽象级别。

4）功能的局部性。公共功能区分出来。

ABSD方法的步骤

（2）架构设计

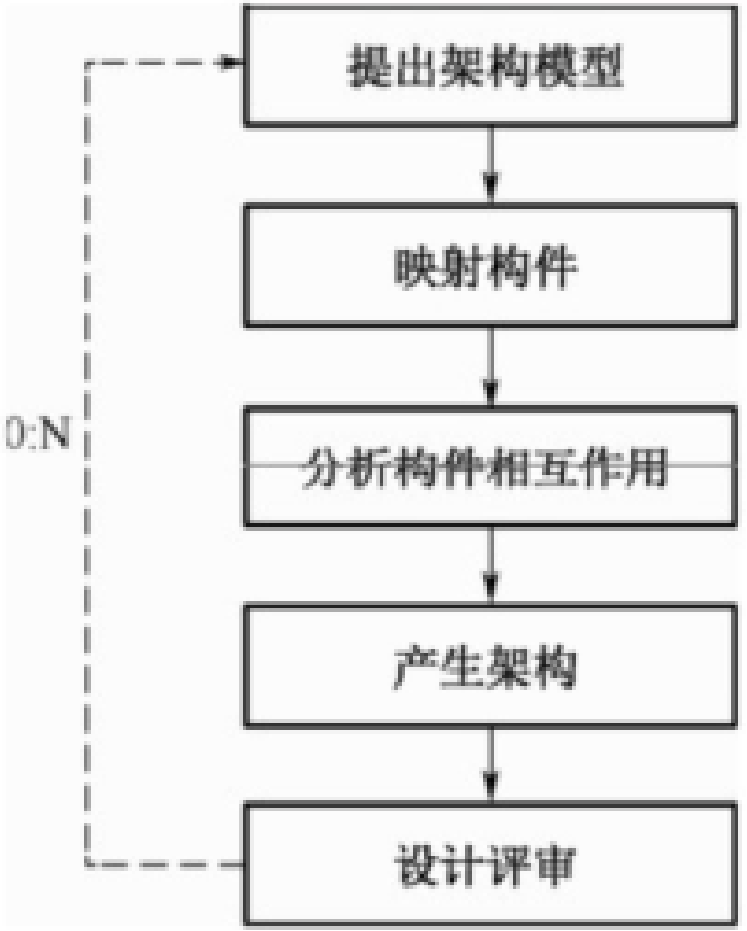


图　　. 架构设计过程

ABSD方法的步骤

（2.1）提出架构模型——选择体系结构风格

- 每个设计元素有一个主要的体系结构风格或模式，这是设计元素如何完成它的功能的基础。主要风格并不是唯一风格，为了达到特定目的，可以进行修改。

- 体系结构风格的选择建立在设计元素的体系结构驱动基础上。

- 在软件设计过程中，并不总是有现成的体系结构风格可供选择为主要的体系结构风格。

ABSD方法的步骤

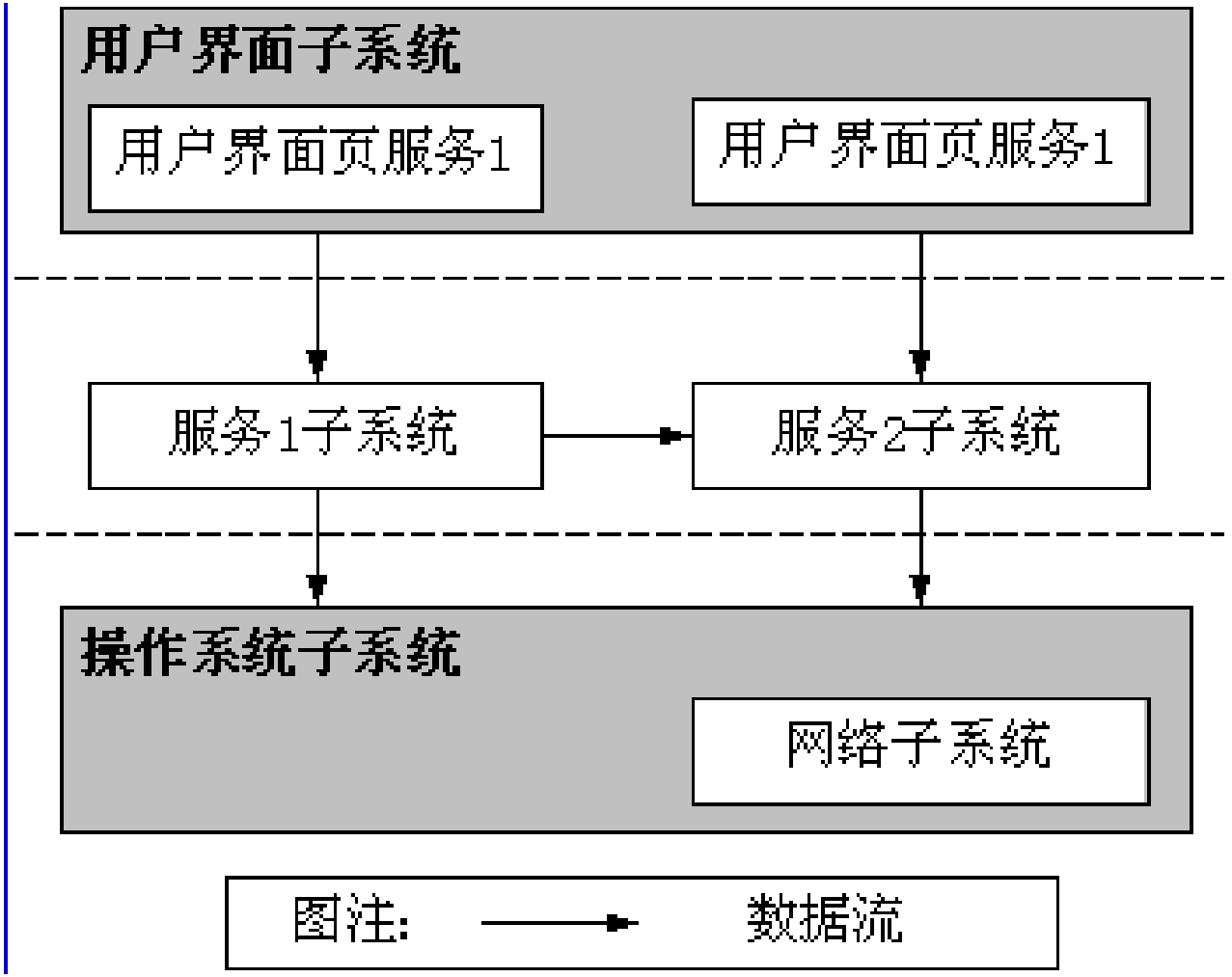
（2.1）提出架构模型——选择体系结构风格

- 一旦选定了一个主要的体系结构风格，该风格必须适应这个设计元素的质量需求，体系结构选择必须满足质量需求。

- 为设计元素选择体系结构风格是一个重要的选择，这种选择在很大程度上依赖于软件设计师的个人设计经验。

ABSD方法的步骤

e. g. 某系统的逻辑视图



ABSD方法的步骤

（2.2）映射构件——为风格分配功能

- 选择体系结构风格时产生了一组构件类型，我们必须决定这些类型的数量和每个类型的功能，这就是分配的目的。在功能分解时产生的功能组，应该分配给选择体系结构风格时产生的构件类型，这包括决定将存在多少个每个构件类型的实例，每个实例将完成什么功能。这样分配后产生的构件将作为设计元素分解的子设计元素。

- 每个设计元素的概念接口也必须得到标识，这个接口包含了设计元素所需的信息和在已经定义了的体系结构风格内的每个构件类型所需要的数据和控制流。

ABSD方法的步骤

（2.3）细化模板，分析构件间关联

- 被分解的设计元素有一组属于它的模板。在ABSD方法的初期，系统没有模板。当模板细化了以后，就要把功能增加上去。这些功能必须由实际构件在设计过程中加以实现。

- 最后，需要检查模板的功能，以判断是否需要增加附加功能到系统任何地方的设计元素中。也就是说，要识别在该级别上已经存在的任何横向服务。模板包括好的设计元素和那些应该共享的功能。每种类型的功能可以根据需要附加支持功能，这种附加功能一旦得到识别，就要进行分配。

ABSD方法的步骤

（2.4）功能校验，设计评审

- 用例用来验证设计元素，验证设计元素是否能够通过一定的结构达到目的。子设计元素的附加功能可以通过用例的使用得到判断。

- 也可以使用变化因素，因为执行一个变化的难点取决于功能的分解。

- 从这种类型的校验出发，设计就是显示需求（通过用例）和支持修改（通过变化因素）。

ABSD方法的步骤

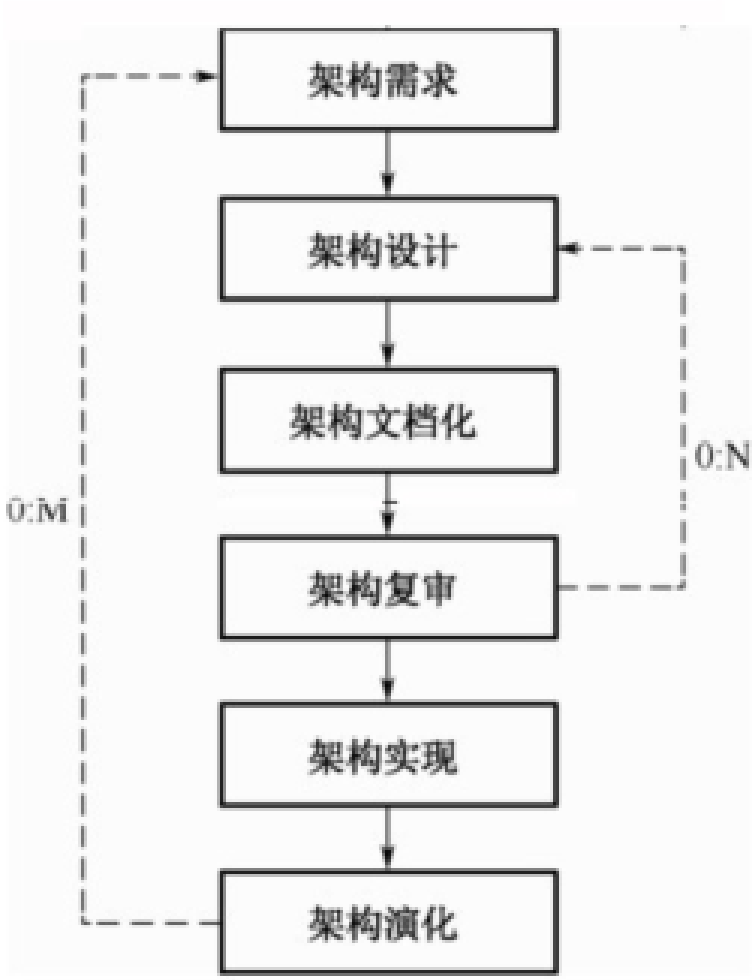


图　　基于架构的软件开发模型

ABSD方法的步骤

（3）架构文档化

- 架构文档化过程的主要输出是架构需求规格说明和测试架构需求的质量设计说明书这两个文档。生成需求模型构件的精确的形式化的描述，作为用户和开发者之间的一个协约。

- 软件架构的文档要求与软件开发项目中的其他文档是类似的。

- 文档要从使用者的角度进行编写，必须分发给所有与系统有关的开发人员，且必须保证开发者手上的文档是最新的。

ABSD方法的步骤

（4）架构复审

- 架构设计、文档化和复审是一个迭代过程。
- 从这个方面来说，在一个主版本的软件架构分析之后，要安排一次由外部人员（用户代表和领域专家）参加的复审。
- 复审的目的是标识潜在的风险，以及尽早发现架构设计中的缺陷和错误，包括架构能否满足需求、质量需求是否在设计中得到体现、层次是否清晰、构件的划分是否合理、文档表达是否明确、构件的设计是否满足功能与性能的要求，等等。

ABSD方法的步骤

（5）架构实现

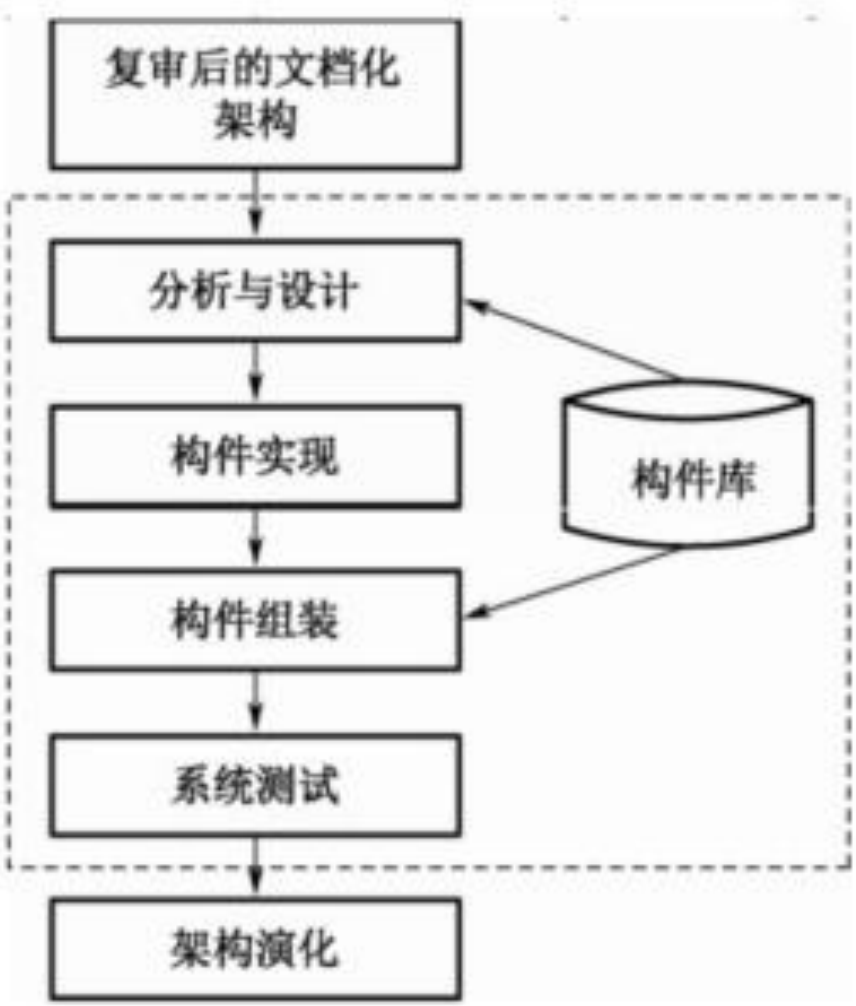


图　　架构实现过程

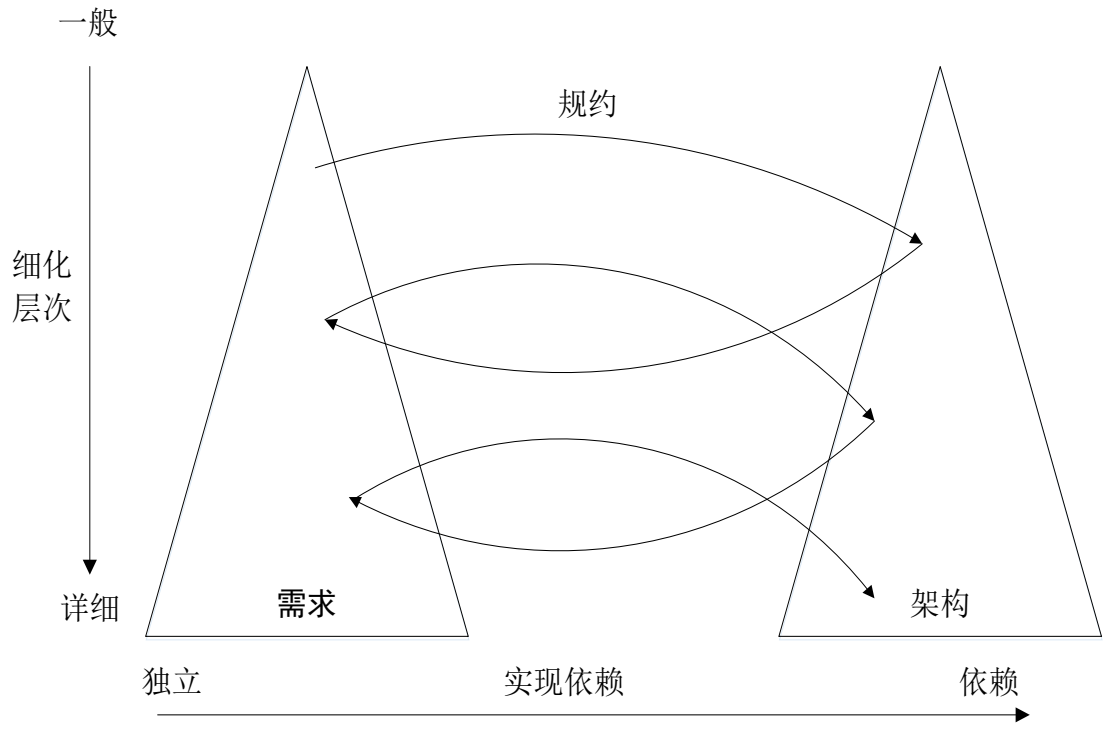
ABSD方法的步骤

（6）架构演化

- 在构件开发过程中，最终用户的需求可能还有变动。哪怕在软件开发完毕，正常运行后，由一个单位移植到另一个单位，需求也会发生变化。在这两种情况下，就必须相应地修改软件架构，以适应新的软件需求。

8.1.3 需求与架构的协同演化

- 软件需求和软件架构两者是相辅相成的关系，一方面软件需求影响软件架构设计，另一方面软件架构帮助需求分析的明确和细化。
- 需求与架构的互相影响可以看成一个螺旋的过程，也是一个双峰的过程。



8.1.3 需求与架构的协同演化

- 双峰模型解决了三个管理问题：
  - （1）I'll Know It When I See It，IKIWISI
    - 需求经常在用户查看模型或原型并提出反馈时才出现。双峰模型允许增量开发，管理开发带来的风险，可以让用户更早地探索解决方案。
  - （2）Commercial off-the-shelf software，COTS
    - 通过双峰模型，开发人员可以从商业可用产品中确定需求及相应的架构，而且可以快速缩小选择范围和确定关键架构决策。
  - （3）Rapid change
    - 分析和识别软件系统核心需求，要求建立一个能适应需求变化的稳定的架构，双峰模型着眼于细粒度发展，容易接受改变。

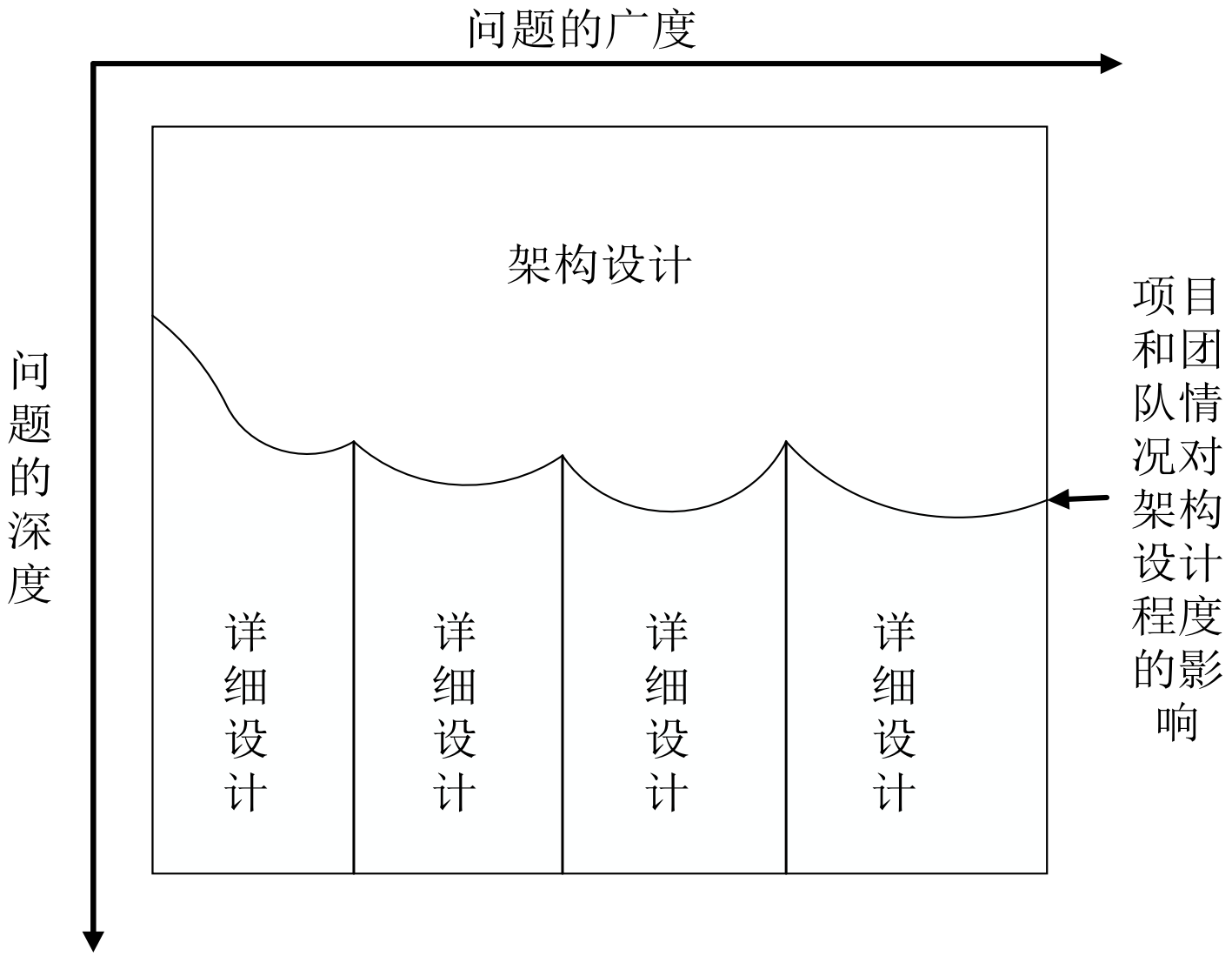
8.1.3 需求与架构的协同演化

- 双峰模型强调软件需求和软件架构的平等性，它是简化版的螺旋模型，在发展需求和架构规约同时，继续从解决方案的结构和规约中分离问题的结构和规约，在一个反复的过程中，产生更详细的需求规约和设计规约，最终把交织在软件开发过程中的设计规约和需求规约分离开来。

8.2从软件架构到详细设计

- 架构设计一般是指关于如何构建软件的一些最重要的设计决策，这些决策往往是围绕将系统分为哪些部分、以及各部分之间如何交互展开的；
- 详细设计是针对每个部分的内部进行设计，是对系统架构设计的精化。
- 软件架构设计应当解决的是全局性的、涉及不同“局部”之间交互的设计问题，一般由软件架构师负责，而不同“局部”的设计由后续的详细设计人员负责。

8.2从软件架构到详细设计



8.2.1详细设计对软件架构的影响

- 详细设计主要集中在架构表达式的细化，选择详细的数据结构和算法。
- 具体的说，详细设计阶段就是确定如何具体实现所需的系统，得到一个接近源代码的软件表示。
- 为了促进从架构向详细设计的转化，可以在软件架构设计中引入详细设计的概念，即在软件架构描述语言中引入与实现相关的元素。
  - 如：把Java语言扩展到支持软件架构描述的ArchJava；在ADL中引入数据类型和函数等支持详细设计

8.2.1详细设计对软件架构的影响

- 如：ArchJava
  - Jonathan Aldrich等人提出，一种新型的软件架构描述语言ArchJava
  - 在Java语言中增加了组件、连接件、端口等建模元素用于描述软件架构模型。
  - 它将软件架构与实现完美的统一起来，确保实现符合架构的限制，支持架构和实现共同开发。

8.2.1详细设计对软件架构的影响

- 再如：研究者在ADL中引入数据类型和函数等支持详细设计
  - Nenad Medvidovic等人将面向对象的类型系统引入到C2中，并可以在设计阶段通过OO 的类、子类型化等概念来规约软件架构建模元素。
  - Dashofy等人提出的xADL2中，允许软件架构设计人员定义与平台或语言(如CORBA，Java等)相关的数据类型、函数声明等

8.2.2从软件架构映射详细设计

- 从设计阶段的软件架构模型向代码的转换过程，是将设计阶段的软件架构模型逐步精化的过程。
- 软件架构映射到详细设计时经常出现的问题：
  - （1）缺失重要架构视图，片面强调功能需求。
  - （2）不够深入，架构设计方案过于笼统，基本还停留在概念性架构的层面，没有提供明确的技术蓝图。
  - （3）名不副实的分层架构，缺失层次之间的交互接口和交互机制，只进行职责划分。
  - （4）在某些方面过度设计。

8.2.2从软件架构映射详细设计

- 可能的解决方法有：
  - （1）对于缺失重要架构视图问题，可以针对遗漏的架构视图进行设计。
  - （2）对于不够深入问题，需要将设计决策细化到和技术相关的层面。
  - （3）对于名不副实的分层架构问题，需要步步深入，明确各层之间的交互接口和交互机制。
  - （4）虽然我们必须考虑到系统的扩展性，可维护性等，但切忌过度设计。

8.2.2从软件架构映射详细设计

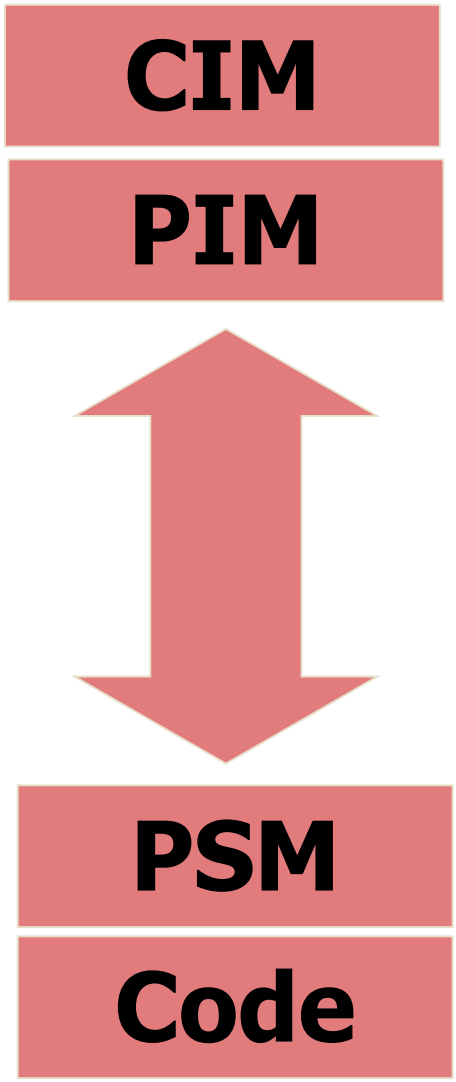
- 从架构描述直接到语言的映射方法：
  - 目前的解决方案或者将高层软件架构模型直接映射成为程序代码，或者经过一系列中间模型的转换，渐进地映射到程序代码。
    - 如：C2 架构描述语言允许将软件架构设计的建模元素映射到面向对象程序设计语言代码，并提供了支持C++和Java的程序库
  - 从ADL向程序代码的映射需要考虑：建立从软件架构建模元素到目标语言元素的映射关系、确保映射过程中的语义正确性、提供自动化的转换工具或环境等问题。
  - 通过直接映射将ADL转化成为程序语言往往只能生成较为简单的程序代码框架。

8.2.2从软件架构映射详细设计

- 基于模型驱动软件架构（MDA， model driven architecture）的映射方法
  - MDA区分了三类模型：
    - 计算无关模型（CIM, Computation Independent Model）：也称业务模型，描述系统的外部行为和运行环境。
    - 平台无关模型（PIM, Platform Independent Model）：具有高抽象层次、无关于任何实现技术的模型。
    - 平台特定模型（PSM, Platform Specific Model）：为某种特定实现技术量身定做，让你用这种技术中可用的实现构造来描述系统的模型。PIM会被变换成一个或多个PSM。

8.2.2从软件架构映射详细设计

- MDA开发步骤
  - （1）用计算无关模型CIM 捕获需求；
  - （2）创建平台无关模型PIM；
  - （3）将PIM转化成为一个或多个平台特定模型PSM，并加入平台特定的规则和代码；
  - （4）将PSM 转化为代码等。



Computation Independent Model

Platform Independent Models

Mappings（映射）：PIM <=> PSM

Platform Specific Models (PSM)

8.2.2从软件架构映射详细设计

- 基于模型驱动软件架构（MDA， model driven architecture）的映射方法
- MDA的根本思想：
  - 将软件系统分成模型和实现两部分：模型是对系统的描述，实现是利用特定技术在特定平台或环境中对模型的解释。模型仅仅负责对系统的描述，与实现技术无关。这是模型的实现技术无关性。

8.2.2从软件架构映射详细设计

- 基于模型驱动软件架构（MDA， model driven architecture）的映射方法
- MDA的好处：
  - 将模型与实现分离后，能够很好的适应技术易变性。由于实现往往高度依赖特定技术和特定平台，当技术发生迁移时，只需针对这种技术作相应的实现，编写相应的运行平台或变换工具。所以，能够比较好的应对实现技术发展带来的挑战。

8.2.3 软件架构视图

- 为了更好地发挥软件架构在系统实现阶段的指导与交流作用，研究者们提出了若干针对实现阶段的软件架构视图
  - Philippe Kruchten于1995年提出4+1视图：逻辑视图、开发视图、进程视图、物理视图、场景视图。
  - Hofmeister Christine提出的四种视图：概念视图、模块视图、执行视图、代码视图。

8.2.3 软件架构视图

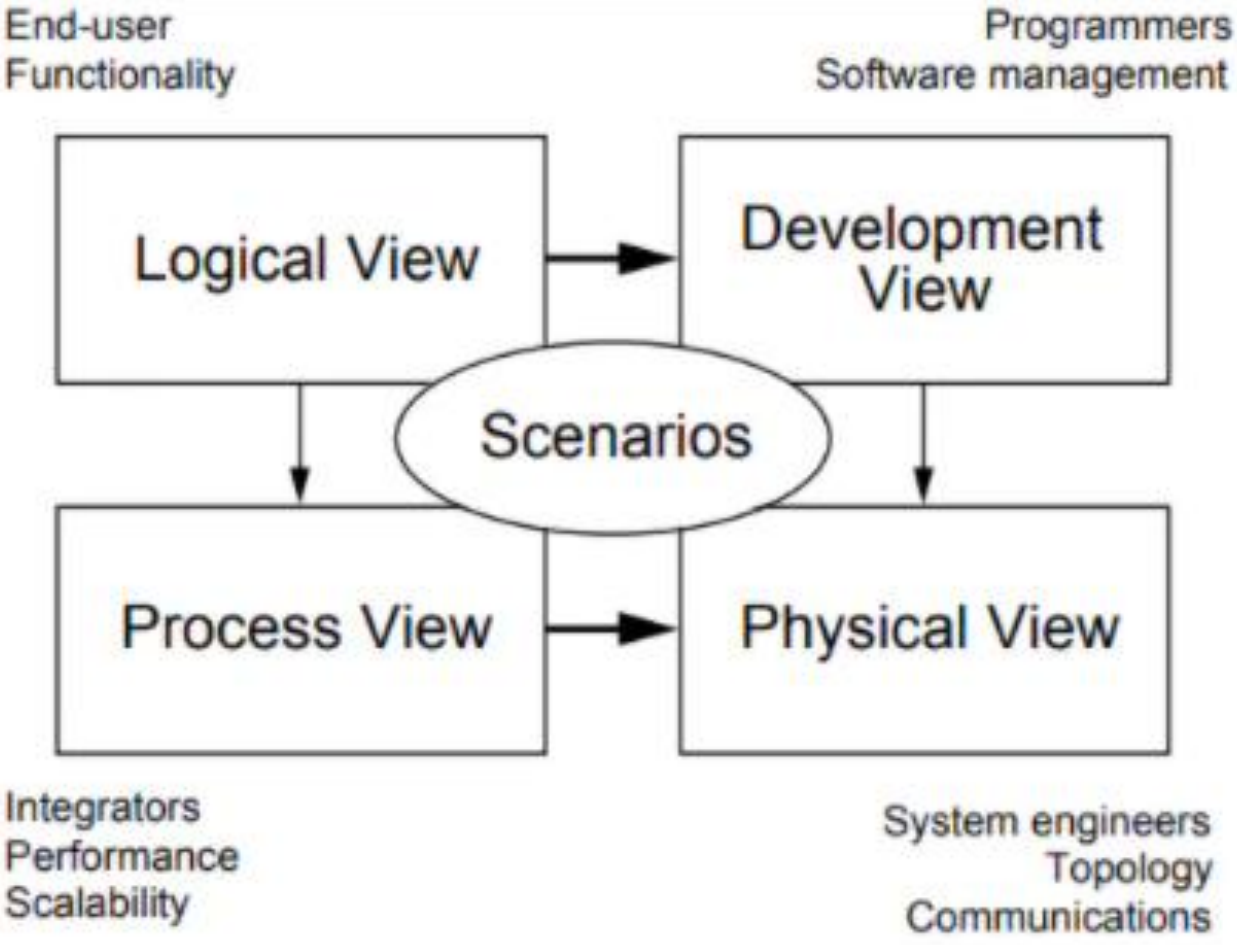


Figure 1 — The "4+1" view model

8.2.3 软件架构视图

- Hofmeister Christine提出的四种视图
  - 概念视图、模块视图、执行视图、代码视图
  - 执行视图描述了模块如何映射到运行时平台（runtime platform）所提供的元素，以及这些又如何映射到硬件架构。
    - （1）全局分析。
    - （2）定义运行时实体。
    - （3）定义运行实体之间预期的、允许的通信路径，这包括通信所使用的机制和资源。
    - （4）通过刻画运行时实体实例的特征和介绍它们的互连方式，来描述系统的运行时间拓扑结构以及这些运行时间实例之间的互联，通过一个执行配置图来描述这一信息。
    - （5）把配置任务中所定义的运行时间实例和预算分配到特定的硬件设备中，同时将特定的值赋给预设的属性（如设置进程的优先级）

8.2.3 软件架构视图

- Hofmeister Christine提出的四种视图
  - 代码视图描述了实现系统的软件是如何被组织的。
    - （1）设计源代码构件。
    - （2）将这些源代码构件组织起来，由开发人员对其进行开发和测试。
    - （3）设计中间构件。
    - （4）设计部署构件。
    - （5）构造过程。设计构造和安装中间构件以及部署构件的过程。
    - （6）配置管理。确定于版本管理和构件发布有关的设计决策。

8.3架构设计原则

- 软件架构设计原则有一般设计原则和关键设计原则两类
- 一般原则
  - 包含商业原则、数据原则、应用程序原则、技术原则等；
- 关键设计原则
  - 关注分离点、单一职责原则、最少知识原则等。

8.3.1 架构设计的基本原则

- 商业原则：
  - 企业利益最大化
  - 信息管理，人人有责
  - 事务持续性
  - 使用通用软件
  - 守法
  - IT责任
  - 知识产权保护

8.3.1 架构设计的基本原则

- 数据原则：
  - 数据资产
  - 数据共享
  - 数据访问
  - 数据托管
  - 使用常用词汇、有数据定义
  - 数据安全

8.3.1 架构设计的基本原则

- 应用程序原则：
  - 技术无关性
  - 易用性
- 技术原则：
  - 需求变化
  - 响应变更管理
  - 控制技术多样性
  - 互操作性

8.3.2架构设计的关键原则

- 关注点分离
  - 将程序分解为相对无关的、功能重叠部分尽可能少的一个个模块，达到高内聚和低耦合的目的。
- 单一职责原则
  - 每个组件或模块应负责一个特定特征或功能，或者内聚功能的集合。
- 最少知识原则(迪米特法则，LoD)
  - 一个组件或对象应该对其他组件或对象的内部细节尽可能少的了解。

8.3.2架构设计的关键原则

- 不重复自身原则 (DRY):
  - 只须在一个地方指定意图，例如，对于程序设计，特定功能只能在一个组件内实现，且该功能不能复制到任何其他组件内。
- 尽量减小前期设计
  - 只做必要的设计。

8.4软件架构设计面临的主要威胁及其对策

- 几个面临威胁的方面：
  - 被忽略的重要非功能需求
  - 频繁变化的需求
  - 考虑不全面的架构设计
  - 不及时的架构验证
  - 较高的创造性的架构比重
  - 架构的低可执行性

8.4软件架构设计面临的主要威胁及其对策

- （1）被忽略的重要非功能需求及其对策
  - 非功能需求大致分为质量属性与约束两大类。
    - 质量属性是软件系统的整体质量品质。如：易用性、性能、可伸缩性、持续可用性、健壮性、安全性、可扩展性、可重用性、可移植性、易理解性和易测试性等都可能是软件的质量属性需求。
    - 约束限制，它们要么是架构设计中必须遵循的限制，要么转化为质量属性需求或者功能需求。

8.4软件架构设计面临的主要威胁及其对策

- （1）被忽略的重要非功能需求及其对策
  - 从开发的角度来看，功能需求的缺陷是可以通过编程级的努力补回，而最终提供的软件系统能否达到非功能需求是无法通过编程级的努力达到的，必须重新架构。
  - 对策：全面认识需求
    - 软件架构强调的是整体，而整体性的设计决策必须给予对需求的全面认识。分层、分类把需求整理清楚，并进行合适的权衡和取舍。

8.4软件架构设计面临的主要威胁及其对策

- （2）频繁变化的需求及其对策：
  - 任何需求变更都可能意味着时间和金钱的消耗，并且大量需求变更之后程序可能变得混乱，势必引起Bug增多，影响产品质量。
  - 对策：关键需求决定架构
    - 在软件架构设计中必须接受变化的普遍存在，采用以关键需求决定架构的策略可以有效应对频繁变化的需求。

8.4软件架构设计面临的主要威胁及其对策

- （3）考虑不全面的架构设计：
  - 软件系统越来越复杂，软件架构也越来越复杂；但软件架构仍须为开发提供足够的指导和限制。

需满足的需求	架构师工作
性能、持续可用性	研究系统运行时情况，合理划分各部分职责，制定并行、分时、队列、缓存等决策
可扩展性、可重用性	研究软件系统开发期间的职责划分、变化隔离、框架使用和代码组织等情况
复杂性	从各方涉众不同角度的关注，使其得以清晰表达

8.4软件架构设计面临的主要威胁及其对策

- （3）考虑不全面的架构设计：
  - 对策：多视图探寻架构
    - 采用分而治之的策略，基于多视图的架构设计方法使在设计过程中无需对所有方面同时进行考虑，而是可以按关注点每次只围绕少数概念和技术展开。

8.4软件架构设计面临的主要威胁及其对策

- （4）不及时的架构验证及其对策
  - 架构设计的合理性直接影响到软件系统最终是否能够成功，因此若未对软件架构合理性进行及早的验证，将为软件开发过程带来巨大的风险。
  - 对策：尽早验证架构
  - 应真正对早期架构进行编码实现，而不仅仅是评审。要对原型进行测试，重点要确保早期架构考虑的质量属性是否能够达到预期。

8.4软件架构设计面临的主要威胁及其对策

- （5）较高的创造性的架构比重及其对策：
  - 在进行软件架构设计的过程中，通常的做法是根据软件项目特征选取某个已有的经典架构作为基础，然后在其基础上根据项目具体要求进行修改调整。
  - 架构的重用也是采用软件架构为软件开发过程所带来巨大好处之一。
  - 但是，对已有架构的理解同样需要时间，可能理解已有架构所耗费的时间比重新设计一个新的架构更多，在这样的场景下，设计新架构貌似是合理的决策。

8.4软件架构设计面临的主要威胁及其对策

- （5）较高的创造性的架构比重及其对策：
  - 已有架构经过实践的检验与修正具有更加稳定的性质，所以在考虑项目时间时引入过多创造性新架构可能为软件开发过程带来不确定性风险。
  - 对策：合理分配经验架构与创新架构比重
  - 为了得到成功的软件架构，需要权衡确定经验架构与创新架构比重的分配。调查显示，80%经验架构加20%创新架构是比较合理的分配比重。

8.4软件架构设计面临的主要威胁及其对策

- （6）架构的低可执行性：
  - 软件架构是从较高层次对软件的抽象，但架构设计的本质是为了保障软件开发过程的顺利进行，所以架构设计并非“好的就是成功的”，而是“适合的才是成功的”。
  - 在架构设计过程中，架构的可执行性应该是贯穿其中的指导思想。

8.4软件架构设计面临的主要威胁及其对策

- （6）架构的低可执行性：
  - 要充分考虑经济型、技术复杂性、发展趋势和团队水平等多方面的因素，制定出合适的架构决策。若一味追求架构的完善性、准确性，而忽视其可执行性，将为软件开发过程带来风险。
  - 对策：验证架构的可执行性
  - 进行架构可执行性验证能够对架构设计过程产生约束效果，从而避免过度设计给软件项目开发带来的风险。

小结

- 在需求分析时如何考虑架构设计
- 如何在需求分析中提炼软件架构
- 软件架构设计如何应对需求变更
- 需求模型到架构设计的映射
- 如何从架构设计出发实现模块设计、算法设计和代码设计

Thanks